The ParaPhrase Project: Parallel Patterns for Adaptive Heterogeneous Multicore Systems

Kevin Hammond[1], Marco Aldinucci[2], Christopher Brown[1], Francesco Cesarini[3], Marco Danelutto[4], Horacio González-Vélez[5], Peter Kilpatrick[6], Rainer Keller[7], Michael Rossbory[8], and Gilad Shainer[9]

[1] School of Computer Science, University of St Andrews, Scotland, UK.
[2] Computer Science Dept., University of Torino, Torino, Italy.
[3] Erlang Solutions Ltd., London, UK.
[4] Dept. Computer Science, Universitá di Pisa, Pisa, Italy.
[5] School of Computing, Robert Gordon University, UK.
[6] School of Electronics, Electrical Eng. and Comp. Sci., Queen's Univ. Belfast, UK.
[7] High Performance Computing Centre, Stuttgart (HLRS), Germany.
[8] Software Competence Centre Hagenberg, Austria.
[9] Senior Director of HPC and Technical Computing, Mellanox Technologies, Israel.

Emails: kh@cs.st-andrews.ac.uk, aldinuc@di.unito.it, chrisb@cs.st-andrews.ac.uk, francesco@erlang-solutions.com, marcod@di.unipi.it, h.gonzalez-velez@rgu.ac.uk, p.kilpatrick@qub.ac.uk, keller@hlrs.de, michael.rossbory@scch.at, Shainer@Mellanox.com.

Abstract. This paper describes the ParaPhrase project, a new 3-year targeted research project funded under **EU Framework 7 Objective 3.4 (Computer Systems)**, starting in October 2011. ParaPhrase aims to follow a new approach to introducing parallelism using advanced refactoring techniques coupled with high-level parallel design patterns. The refactoring approach will use these design patterns to restructure programs defined as networks of software components into other forms that are more suited to parallel execution. The programmer will be aided by high-level cost information that will be integrated into the refactoring tools. The implementation of these patterns will then use a well-understood algorithmic skeleton approach to achieve good parallelism.
A key ParaPhrase design goal is that parallel components are intended to match *heterogeneous* architectures, defined in terms of CPU/GPU combinations, for example. In order to achieve this, the ParaPhrase approach will map components at link time to the available hardware, and will then re-map them during program execution, taking account of multiple applications, changes in hardware resource availability, the desire to reduce communication costs etc. In this way, we aim to develop a new approach to programming that will be able to produce software that can adapt to dynamic changes in the system environment. Moreover, by using a strong component basis for parallelism, we can achieve potentially significant gains in terms of reducing sharing at a high level of abstraction, and so in reducing or even eliminating the costs that are usually associated with cache management, locking, and synchronisation.

1 Introduction

From the 1960s until very recently, hardware designers were able to exploit the effects of Moore's law to create processors with ever-increasing clock frequencies. Software benefited from each new processor generation more or less automatically. At the same time, software engineers remained essentially wedded to the inherently sequential *von Neumann* programming model that has been in use ever since the early days of computing. Most of the major advances in programming language technology and software engineering that have taken place (e.g. structured programming, object-orientation, or abstract modelling) were therefore solely motivated mainly by the need to keep ever-larger software systems manageable, rather than to make effective use of the available hardware capabilities. This situation is currently changing, however, *and changing extremely rapidly*. Future multicore/manycore hardware will not be *slightly parallel*, like today's dual-core and quad-core processor architectures, but will be *massively parallel*. Concurrently with this trend towards increasing numbers of cores, there is also a strong trend towards *heterogeneous* architectures, with chips containing not only conventional processor cores, but also various specialist processing units such as graphics-processing units (GPUs), physics engines, digital signal processors (DSPs), etc. Properly exploiting heterogeneous multicore technology is essential for today's users of high-performance computers: provided they can be properly harnessed, hybrid multicore/manycore systems offer the potential for cheap, scalable and energy-efficient high-performance computing. Unfortunately, while GPU computing [52] compares very favourably with multicore CPUs in terms of performance, *it has even worse programmability*. It is therefore becoming increasingly obvious that the traditional sequential programming model has reached its limits. This problem of programmability for future parallel computers motivates the ParaPhrase project.

The Challenge. It is clear that effectively exploiting *heterogeneous* multicore/manycore processor technology will be an essential requirement for future software developers. The main challenge they face is finding a programming model that provides a suitable level of abstraction, while still allowing good use of the available hardware resources. It is already very difficult for classically-trained applications programmers to benefit from the performance offered by today's multicore systems, and only highly-skilled programmers or those seeking the highest levels of performance are presently exposed to parallel programming techniques [1]. Without a fundamental shift in the programming model, programmers will find it essentially impossible to exploit the mid-term/long-term developments that major hardware companies such as Intel and NVidia promise to deliver. The dilemma is that a large percentage of mission-critical enterprise applications will not "automagically" run faster on multicore servers. In fact, many will actually run slower [43]. It is therefore essential that we make it as easy as possible for applications programmers to exploit the latest developments in heterogeneous multicore/manycore architectures, while still making it easy to target future (and perhaps unanticipated) hardware developments.

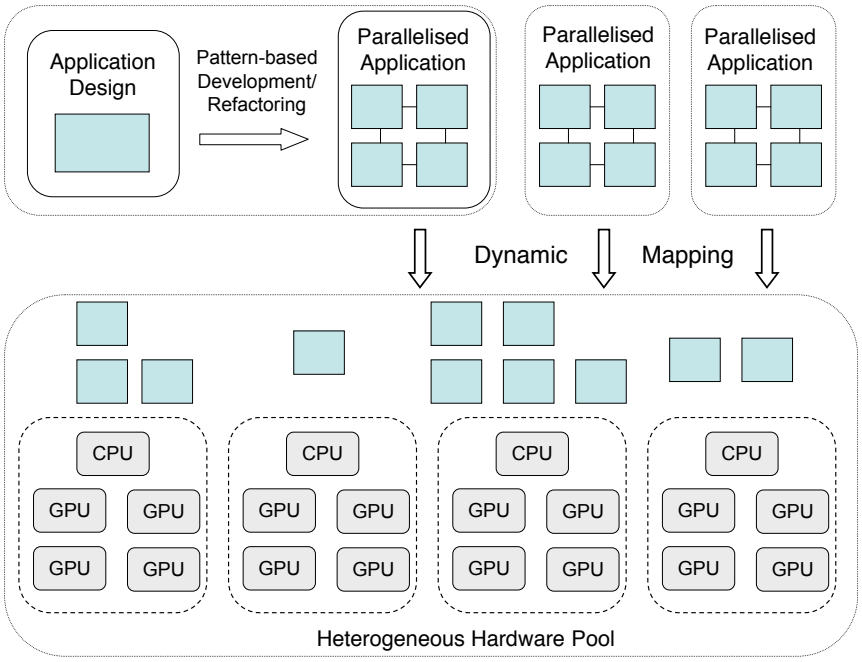

Fig. 1. The ParaPhrase Vision

2 Related Work

Pattern-Based Parallel Programming Models. The recent advent of multicore processors, GPUs, chip multiprocessors, and multinode clusters and constellations has dramatically increased the number of concurrent processors that are available within a single system configuration. Architectures involving dozens of heterogeneous core in an integrated processing node are becoming commonplace in high-performance computing environments, and, as a result, state-of-the-art supercomputing facilities must efficiently administrate thousands of processing units. JUGENE (the Blue Gene/P PRACE platform at Julich) features 294,912 "traditional" multicore processing elements and the Tianhe-1 comprises 186,368 CPU/GPUs, according to the November 2010 Top500 list [60]. However, software development techniques have not evolved at the same pace and the software itself has typically outlived architectural generations. Linpack, the canonical HPC benchmark, was initially released in 1979 and still constitutes the basis for the Top500 list. While the *linguae franca* for large parallel computers have historically been Fortran and C coupled with coordination libraries such as MPI, OpenMP, or PVM, there is no clear trend on how to efficiently engineer the required parallel programs for mainstream parallel computers. Research effort needs to be devoted to design efficient parallel programs that not only exploit the architectural characteristics of parallel architectures, but also preserve the investment in programming over time in a number of platforms. The challenge is

therefore to holistically develop high-quality parallel software, which is not only efficient and scalable, but which is also well-programmed and generic.

While they were originally defined as abstractions of common themes in object-oriented programming [25, 26], design patterns have subsequently been incorporated into parallel programming methodologies [50]. Pattern-based parallel programming allows an application programmer the freedom to generate new parallel programs by parameterising parallel abstractions. This approach is very similar to that taken by algorithmic skeletons, described below. The main differences lie in their respective visions [18]: in the design pattern approach the (software engineering driven) vision proposes patterns as *models* to be instantiated, implemented and used by the programmer; in the algorithmic skeleton approach the (language-driven) vision uses skeletons as predefined *language constructs* or library entries that can be seamlessly used by the programmer in the same way as other non-parallel language constructs and/or library entries.

The parallel programming pattern concept has been extended into a design method under the umbrella of *parallel pattern languages*. Unlike other parallel programming languages, parallel pattern languages present rules for designing parallel programs based on problem-class abstractions which describe parallel structure, dataflow, and communication; critical region locks, such as test-and-set and queued for simple mutual exclusion, or reader/writer for concurrent execution; or socket-based operators for web applications.

In their frequently-cited report, Asanovic et al. have emphatically suggested the deployment of parallel design patterns to successfully produce effective parallel programs for multi and manycore architectures [7]. However, the generic implementation of these parallel design patterns in multi and manycore architectures requires the use of refined techniques which can provide a clear-cut separation of software and hardware. This has long been considered critical to the success of any parallel programming endeavour since it is essential if we are to foster the reuse of algorithms and software. Moreover, we consider that the division of the structure from the application itself is crucial to the goal of delivering adaptability.

Algorithmic Skeletons. *Algorithmic skeletons* abstract commonly-used patterns of parallel computation, communication, and interaction into a set of language constructs [17, 32]. Skeletons present a top-down structured approach where parallel programs are formed from the parametrisation of skeleton nest, also known as structured parallelism. Structured parallelism deployed through skeleton frameworks provides a clear and consistent structure across platforms by distinctly decoupling the application from the structure in a parallel program. It does not rely on any specific hardware and it benefits entirely from any performance improvements in the system infrastructure. Algorithmic skeleton frameworks (provided either as new languages or as libraries) have been implemented using a variety of techniques including macro data flow, templates, aspect-oriented programming, and rewriting techniques to target distributed architectures (such as COW/NOWs and grids) and, more recently, homogeneous

multicore architectures. Recently, implementation techniques supporting *expandable* algorithmic skeleton sets have been demonstrated [3]. It is arguable that the different techniques used to implement algorithmic skeleton frameworks are suitable to support implementations targeting heterogeneous multicore architectures. While algorithmic skeletons (and indeed parallel patterns in general) cannot be used to produce *all* parallel and distributed programs, there is a growing number of important applications [19, 55]. A number of recent and highly successful "programming models" such as the well-known *Google MapReduce* also derive and inherit from algorithmic skeletons [13]. Furthermore, skeletal methodologies inherently possess a predictable communication and computation structure, since they directly capture the structure of the program. They therefore provide, by construction, a foundation for performance modelling and estimation of parallel applications.

Refactoring Technology for Parallel Programming. Refactoring [51] changes a program's structure, but keeps its behaviour the same. *Software refactoring* involves using tool support to adapt or change existing software according to well-defined patterns. It is primarily used to produce code that is either more efficient, that uses specific library/language capabilities, or that is better structured to meet some software engineering goals. The primary challenges lie in: i) identifying the refactorings that are available to the programmer; ii) guiding the programmer in determining which of those refactorings are most sensible/beneficial; and iii) correctly applying the refactoring so that the resulting code has the required behaviour without introducing unwanted changes in functionality. Refactoring tools such as Eclipse [24] now offer an extensive range of refactorings also including inlining, extract constant, introduce parameter and encapsulate a field. Many refactoring tools are fully-fledged commercial or open-source products.

Refactoring Parallel Programs. Despite the obvious advantages, there has so far been little work in the field of applying software refactoring technology to assist parallel programming. The earliest work on interactive tools for parallelisation stemmed from the Fortran community, targeting loop parallelisation [40]. These interactive tools were early transformation engines allowing users to manipulate loops in their Fortran programs by specifying what loops to interchange, align, replicate or expand. The interactive tools typically reported to the programmer various information such as dependance graphs, and was mainly applied to the field of numerical computation. Recent work in the field includes Reentrancer [62]: a refactoring tool developed by IBM for making code reentrant. Reentrancer targets global data by making them thread-safe. Further recent work includes a refactoring approach to parallelism by Dig [20], targeted at introducing concurrency in Java programs by aiming to make them more thread safe, increasing throughput and scalability. Hitherto, Dig's refactoring tool contains a minor selection of transformations including *make class immutable*, *parallelise loop* and *convert HashMap to ConcurrentHashMap*. Software refactoring

techniques have therefore only previously been applied in a very limited parallel setting: by applying simple transformations to introduce parallel loops and thread safety in object-oriented (OO) programs. Currently, these approaches do not take any extra function properties into account, such as hardware characteristics, costing and profiling, for aiding the refactoring process. Furthermore, the techniques are rather limited to homogeneous architectures and OO languages, rather than applying general patterns to heterogeneous architectures, as needed in the ParaPhrase project.

Automatic Parallelisation. (Semi-)Automatic parallelisation poses two main challenges: i) how to identify those parts of the program that could be executed concurrently; and ii) how to map these parts onto a given set of computing resources. Failing in either challenge immediately limits any potential performance gains. Both of these challenges are clearly addressed within the ParaPhrase project. Most research on automatic parallelisation has focused on how to identify concurrency. Many sophisticated optimisation techniques based on dependence analyses have been developed [8, 63, 6]. They form the basis for the *polytope model* [42, 23, 9], which facilitates compiler-directed transformations to increase loop-level concurrency. Such approaches are fundamentally limited, however, to specific programming patterns, and tend to favour fine-grained parallelism. In the ParaPhrase project, we take a higher-level approach to identifying parallelism, recognising that programmer assistance and insight may be valuable at this stage. This has three main advantages: firstly, the pattern-based approach allows us to easily decompose the application into suitably concurrent tasks, *breaking accidental dependencies that will limit the polytope approach*; secondly, we can use performance information not only to drive the choice of parallel implementation, as commonly happens, but also to guide programmer-directed refactoring to identify the most profitable parallel structure; and thirdly, by using a component structure with a strong explicit resource interface that automatically exposes necessary inter-task dependencies, and avoids accidental dependencies that restrict the opportunities for concurrency. Having identified good parallelism, it is necessary to focus on the issues involved in the second challenge, i.e. *effectively* mapping concurrency onto the available computing resources so as to maximise performance. This mapping requires decisions to be made such as: Which concurrently executable parts should actually be done in parallel? How and when should synchronisation happen? Where should data be placed? What layout in memory should be used for the data to enable non-conflicting (and cache-friendly) concurrent access? etc.

A large body of work addresses these issues in the context of nested loops. Besides scheduling-driven [22, 38] and partitioning-driven [44, 45] approaches, more recent tiling-based [12] and streamisation-based [53] approaches have shown promising results for shared-memory architectures. Any compiler-driven decision mechanism for these aspects relies on the availability of as precise as possible knowledge of application properties such as typical data sizes, function application frequencies, typical parameter ranges etc. Over recent years we have devel-

oped several analysis and program transformation techniques that aim at identifying these properties. Amongst these are partial evaluation techniques such as those described in [36, 59, 10, 39], as well as code restructuring techniques [56, 34, 37], and multi-threaded code generation techniques [33, 35]. However, the interplay of such optimisations combined with the complexity and variety of target platforms often renders static mapping approaches far less effective than the mappings that can easily be achieved manually. Even in the single-processor setting it has been shown that semi-static approaches such as iterative optimisations [47, 54] can improve the effectiveness of the optimisation process. The ParaPhrase approach avoids these problems by exploiting a more dynamic approach that can adapt to changing system conditions, given a good initial placement as its starting point.

Hardware/Software Virtualisation In the context of ParaPhrase, the purpose of the hardware/software virtualisation layers is: i) to abstract over the available heterogeneous multicore hardware in order to support the automated mapping of an application onto diverse targets; ii) to support dynamic remapping and adaptivity; and iii) to support the seamless mapping of multiple simultaneous parallel applications to the available hardware resources. This represents a significant challenge. The virtualisation must allow the decomposition of the parallel software into units that can easily be mapped/re-mapped to alternative hardware realisations; it must support cost information that allows rapid decisions to made on dynamic re-mapping; it must be sufficiently flexible that it can support all the required parallel patterns; and it must be sufficiently lightweight that it does not impose excessive overhead that may restrict the flexibility of dynamic re-mapping.

The state-of-the-art in dynamic targeting is epitomised by the Java Virtual Machine (JVM) [46], where compiled code, represented as an abstract instruction set (Java byte-code), is either interpreted by the JVM or is compiled on execution into the instruction set of that processor. Virtualisation can also be used to translate between instruction sets. For example, the full virtualisation of a target is possible where a virtual machine environment is able to execute all software that is capable of executing on that target. A good example is the Transmeta Crusoe architecture [27], which provides a full virtualisation of the x86 platform onto a much more energy-efficient VLIW processor. The techniques exploited here use a mixture of binary translation from native x86 binaries to the Crusoe's VLIW instruction set together with a mechanism for caching recently translated code blocks. These techniques, namely interpretation, binary translation and just-in-time compilation may be augmented with the use of fat binaries, where the choice of target or target parameterisation is reasonably bounded. All of these techniques could be exploited by ParaPhrase. However, the main issue for ParaPhrase is the efficient execution of generic parallel code on an arbitrary heterogeneous target. Although the Java execution model supports concurrency, this model was originally designed to support threaded programs on a single processor rather than supporting distributed parallel programming. Moreover,

the model is not constrained, which means that the programmer must ensure both that threads do not interfere with each other and that resources are properly synchronised. It therefore does not meet the objectives posed by ParaPhrase. It follows that a more general model of concurrency is required, one that ideally is safely composable without inducing deadlock or compromising efficiency. By using a new virtualisation model based around costable software components coupled with a simple hardware virtualisation, we anticipate that we will be able to meet the stringent requirements of the ParaPhrase project.

Autonomic and Dynamic Placement Placement of concurrent components derived from the compilation of high level parallel patterns on multicore, heterogenous architectures poses different problems related to efficiency and performance. Vadhiyar and Dongarra [61] suggest that a "self-adaptive software system examines the characteristics of the computing environments and chooses the software parameters needed to achieve high efficiency on that environment". Thus, we consider that the key challenges in adaptively improving the performance of parallel programs in a heterogeneous system are therefore:

1. the correct selection of resources (processors, links) from those available;
2. the correct adjustment of algorithmic parameters (for example, blocking of communications, granularity); and, most importantly,
3. the ability to adjust all of these factors *dynamically* in the light of evolving external pressure on the chosen resources.

Although different parallel solutions for heterogeneous distributed systems have traditionally exhibited parallel patterns, their associated optimisations have not necessarily exploited the application structure. They have either modified the scheduler [14] or kept the actual application interlaced [57], without decoupling the structure from the behaviour. Such challenges are aligned with the traditional view on intra-application scheduling, which proposes five actions: i) select resources to schedule the tasks; ii) map tasks to processors; iii) distribute data; iv) order tasks on compute resources; and, v) order communication tasks. However, traditional strategies for placement or scheduling in distributed systems [11, 15, 21, 41, 49] rely on system simulators, dedicated configurations, and/or performance estimators to model the general system, particularly to characterise the background load in terms of its job arrival rate. While much can be said about the reproducibility of their results, one may argue that they artificially create tractable evaluation scenarios for their scheduling policies. It follows that he ParaPhrase methodology cannot be simplistically compared to any task scheduling policy in terms of algorithmic optimality and complexity, but ought to be evaluated in terms of the makespan for a certain workload.

In addition to the preliminary, possibly static, optimisations performed when deploying the program components onto the target architecture, based on the expected performance models of the parallel patterns used, complementary approaches will be considered:

Control loops. This approach extends the experience of the Universities of Pisa and Torino in Behavioural Skeletons [5, 2, 4].

Divisible Workloads. This approach builds on previous work from several partners including that of Robert Gordon University on statistical scheduling of divisible workloads [30] and the systematic introduction of adaptivity into parallel patterns and skeletons [29, 31, 4].

Each of the components derived from the compilation of the high-level patterns will be equipped with a couple of additional interfaces: a *sensor* interface and an *actuator* interface. The former will provide methods suitable to gather actual measures related to the current status of the computation (e.g. throughput, service time, latency). The latter will provide methods to *modify* the implementation of the high level pattern (e.g. change its parallelism degree by adding/removing components, migrating the component from CPUs to GPUs (or *vice-versa*). An additional *autonomic performance manager* component will be added to the implementation of each high level parallel pattern. The manager will implement a control loop. Performance of the parallel pattern implementation will be monitored through the sensor interface and possibly actions performed by invoking the actuator methods to improve overall pattern performance. The autonomic manager may be implemented on top of a business rule engine in such a way the rules executed at each control loop iteration may embody all techniques to dynamically optimise the performance of the high level patterns as well as any new and/or experimental techniques. In particular, techniques based on learning from previous experience may be implemented, provided a data base of past computation management is maintained.

In summary, the ParaPhrase approach can be categorised as a autonomic and dynamic placement methodology for parallel programs executing in heterogeneous distributed systems, which is:

dynamic since the correct selection of resources and the adjustment of algorithmic parameters are performed at execution time;

autonomic due to the provision of intrinsic mechanisms to dynamically adjust to variations in system performance;

application-level because all decisions are based on the specific requirements of the application at hand; and,

heuristic because it comprises a set of rules intended to increase the probability of enhancing the overall parallel program performance.

3 The ParaPhrase Project

The challenges identified in Section 1 require a new and radical approach that tackles parallel programming in a coherent and holistic way. The ParaPhrase project aims to produce a new structured design and implementation process for heterogeneous parallel architectures, where developers exploit a variety of parallel patterns to develop component-based applications that can be mapped to the available hardware resources, and which may then be dynamically re-mapped to

meet application needs and hardware availability (Figure 1). We will exploit new developments in the implementation of parallel patterns that will allow us to express a variety of parallel algorithms as compositions of lightweight software components forming a collection of virtual parallel tasks. Components from multiple applications will be instantiated and dynamically allocated to the available hardware resources through a simple and efficient software virtualisation layer. In this way, we will promote adaptivity, *not only at an application level, but also at a system level.* Finally, virtualisation abstractions will be provided across the hardware boundaries, allowing components to be dynamically re-mapped to either CPU or GPU resources on the basis of suitability and availability.

3.1 Achieving the ParaPhrase Project Vision

In order to achieve the vision described above so that we can make effective use of recent and future advances in heterogeneous multicore/manycore computing, a number of key technical problems must be addressed.

We must develop new parallel programming models. The parallel programming models in widespread use today require the programmer to manage many low-level organisational details, including communication, placement and synchronisation. This low-level coding style makes programming parallel systems notoriously difficult, since a whole new class of programming errors severely impacts programming productivity as mismatched communications, deadlocks, race conditions, which often exhibit non-deterministic behaviour

We must develop new means of identifying parallelism. Low-level parallelism libraries, such as OpenMP, MPI or Pthreads are widely used in non-numerical applications. However, such approaches are highly inflexible, making it hard: to dynamically adapt to the execution environment; to introduce the high-level changes to program structure that may be necessary to support new multicore computer architectures; or to refactor existing program code to support a new parallel application. What is needed is a simpler way of identifying parallelism that can both support adaptation to conform to dynamic changes in hardware availability and that can support long-term software evolution to meet new application or hardware needs.

We must develop new ways of abstracting across the capabilities of different architectures/devices. Code written for a general-purpose GPU (GPGPU) cannot easily be executed on a general-purpose CPU core, or *vice-versa*. This limits the ability to reconfigure software to exploit the available hardware resources, effectively restricting software to a static placement.

The ParaPhrase project focuses on issues of programmability for parallel systems. It will develop a new approach based on patterns of parallelism that structure independent parallel components. Each independent component will have a well-defined interface identifying memory dependencies and key extra-functional properties, such as performance information, degree of parallelism and communication access patterns. The patterns and components will be identified using

a high-level and novel *refactoring* approach that will guide the programmer towards optimal design decisions targeting a range of different implementations. Having chosen a pattern, the pattern must be *mapped* to the available hardware targets using a *behavioural skeleton* approach. Finally, since it is well known that even an optimal initial placement is unlikely to achieve maximum performance under real-world operating conditions, components from multiple applications will be *re-mapped* during execution in order to maximise performance. The project thus combines automatic approaches to task placement and re-mapping with an assisted approach to the initial identification of the parallel program structure. The use of a component-based approach, with strong behavioural interfaces forming a virtual parallel task structure, is fundamental in allowing the ParaPhrase project to achieve its goals.

3.2 Key Technologies

The key technologies used in and deployed by the ParaPhrase project are:

Refactoring: the process of changing the structure of a program without changing its behaviour. Refactoring has been practised implicitly for as long as programs have been written, as programmers actively re-structure their code as they typically build software. Refactoring tools, such as the Paraphrase Refactoring Tool, will provide a set of well-defined semi-automatic refactorings aimed at parallelisation that will allow the programmer to simply select which parallel pattern (or skeleton) to apply. The refactoring tool then checks any conditions and applies the transformations automatically.

Virtualisation: the ParaPhrase project deploys two levels of virtualisation. *Component Virtualisation* abstracts over different software implementations of the same parallel program, allowing parallel programs to be composed from several software components. *Hardware Virtualisation* abstracts over the heterogeneous hardware resources that are available, allowing software components to be mapped/re-mapped to alternative physical hardware.

Parallel Patterns: high-level expressions of generalised parallel algorithms that typify *classes* of parallel problems. Typical parallel patterns include task farms, work pools, pipelines, parallel maps and parallel reduce operations.

Skeletons: "implementations" of design patterns. A skeleton provides a parametric implementation of a specific parallel design pattern targeting a given class of architecture.

FastFlow: a skeleton-based programming framework, developed at the Universities of Pisa and Torino, that efficiently targets cache-coherent multicore architectures.

Erlang: a strict functional language with dynamic types and support for built-in concurrency. Erlang is executed on the Erlang Virtual Machine layer, which provides implicit mechanisms for managing fault tolerance and for deploying data serialisation in message passing.

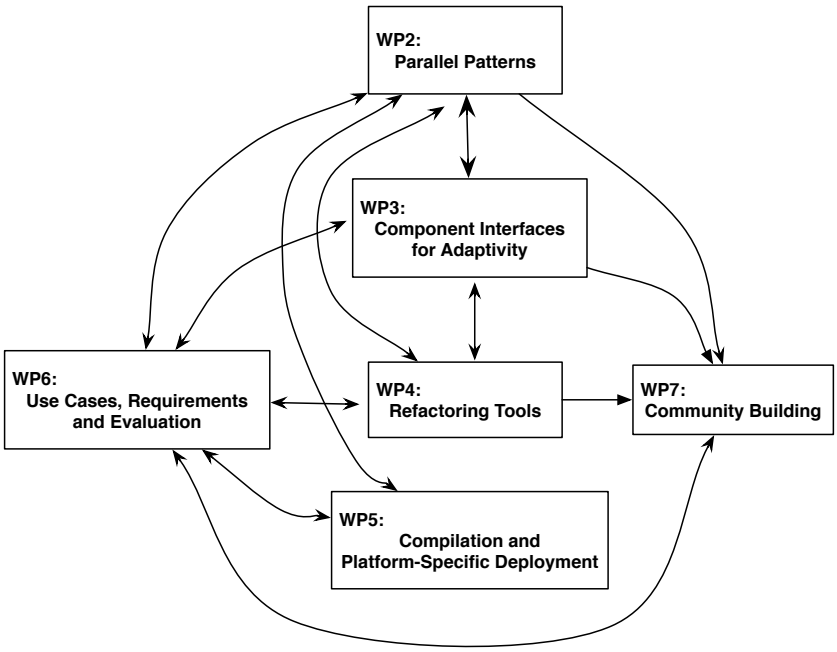

Fig. 2. The ParaPhrase Workpackages and their Dependencies

3.3 ParaPhrase Structure and Workplan

An outline structure of the ParaPhrase project into its component technical workpackages (WP2–WP7; WP1 is Management) is shown in Figure 2. WP2 covers high-level parallel patterns and their implementation as skeletons. WP3 defines the software virtualisation framework. WP4 develops new refactoring tools and techniques. WP5 considers adaptive mapping technology. WP6 validates the work done in the project against some real applications. Finally, WP7 aims to develop a user community for the ParaPhrase technologies.

High-Level Parallel Patterns (WP2). The use of a pattern-based approach allows parallelism to be expressed at a very high level of abstraction, so achieving the overall aim of simplifying the programming of multicore systems. At the same time, by exposing parallelism in terms of specific parallel patterns, parallel programs can be easily refactored into alternative forms with different parallel behaviours, as indicated in Figure 3. For example, a parallel *map* operation, where a single operation is applied in parallel to every element of a collection of data may be either refactored into a parallel *task farm*, with a fixed mapping of tasks to processing elements; or may alternatively be refactored to a parallel *workpool*, where the tasks are mapped dynamically to processing elements as

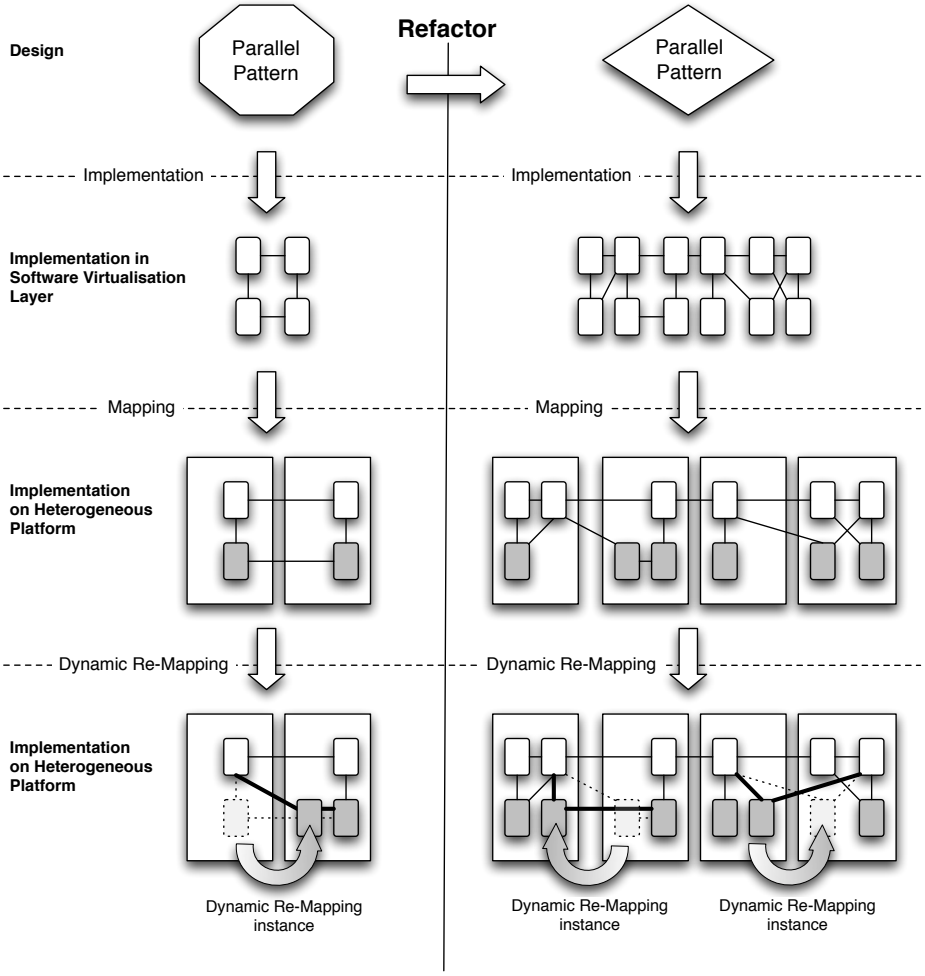

Fig. 3. The ParaPhrase Approach: Refactoring and Implementing Parallel Patterns

they become available. Depending on the structure of the parallel application, one or other of these patterns may be preferable: for example, a task farm is more suitable for more regular task sizes; where a workpool is more suitable for irregular task sizes. Achieving this objective will therefore allow us to improve programmability of multicore systems.

Heterogeneous Pattern Implementations (WP2). Heterogeneity and parallelism are critical to future high-performance computers: future computer architectures are likely to be built around collections of large numbers of parallel processing elements, comprising both general-purpose units (CPUs), but also higher-performance but more specialised units (e.g. GPUs, DSPs, Physics

Engines, FPGAs, ASICs etc). These units may have overlapping, but not interchangeable capabilities. These units may be grouped into different configurations, comprising different ratios of general-purpose to special-purpose units, different clock speeds etc. In order to make effective use of the available processing elements in such an architecture, it is therefore essential to consider heterogeneity. Achieving this objective will thus allow us to demonstrate the benefits of using high-level patterns for heterogeneous multicore systems in future high-performance computing applications.

Software Virtualisation Framework (WP3). Once a parallel program has been refactored into the required parallel pattern, it can then be decomposed into a set of cooperating parallel components, with well-defined communication interfaces, and interconnections using an *algorithmic skeleton* approach (Section 2). This componentisation and encapsulation is important, since by providing alternative implementations of a component, that component can be mapped to different kinds of hardware processing elements, for example to either a CPU or a GPU. Since the use of high-level patterns will allow programs to be decomposed into potentially large numbers of parallel components, and we will be able to use the same hardware to execute components from multiple parallel applications, we will have a great deal of flexibility both when initially mapping components to CPU/GPU elements, and in subsequently re-mapping components to CPU/GPU cores as a result of dynamic changes in the execution environment.

Refactoring Tools for Parallel Patterns (WP4). Refactoring tools support programmer-directed transformation of the source code in order to improve behavioural or other properties of program code. In the ParaPhrase project, we are interested in *supporting* the programmer by allowing them to choose between alternative parallel patterns, using high-level information about their runtime behaviours. While it would be, in principle, possible to automatically map high-level programming patterns directly to implementations, as has previously been done for some *algorithmic skeletons*, such an approach requires excellent cost modelling, *and usually restricts the choice of implementation.* By using a refactoring approach, much of this machinery and the associated complexity can be avoided in favour of a programmer-directed system. The refactoring technology will also allow us to automatically insert appropriate component interfaces, including extra-functional (behavioural) information. In this way, we will help to achieve our overall aim of reducing the complexity of identifying parallelism for heterogeneous multicore systems.

Adaptive Mapping Technology (WP5). We need to develop methods to map software components onto the resources of a heterogeneous multicore platform, matching them against the available hardware characterisation that is exposed through the hardware/software virtualisation layers. This mapping needs to take into account both computations, which will mapped to the available

hardware resources, and any communication that is induced by this mapping. In this way, we will achieve our aim of developing new dynamic mechanisms to support adaptivity and heterogeneity.

Application-Based Validation (WP6). We have already identified a number of target high-performance applications from the data analysis, machine learning and weather prediction domains. These applications must demonstrate good multicore performance and expose opportunities for heterogeneity. They will be used to study the effectiveness of the various stages of our approach, including that the mapping and placement technology improves performance both in an initial placement onto a heterogeneous multicore application, and through system reconfiguration during execution.

User Community Building (WP7). ParaPhrase aims to create a user community to ensure longer-term uptake of the technologies developed in the project. Driven by the consortium industrial partners, this community will encompass a multiplicity of stakeholders exploiting close connections with the HPC Advisory Council (http://www.hpcadvisorycouncil.com/, "a computing ecosystem that includes best-in-class original equipment manufacturers (OEMs), strategic technology suppliers, independent software vendors (ISVs) and selected end-users across the entire range of HPC market segments."

3.4 ParaPhrase Use-Cases

The practical utilizability of the ParaPhrase approach especially concerning simplification of parallel development and performance gain will be demonstrated using real applications from industrial, scientific and video streaming domains. The focus on industrial applications for example is highly relevant in practice due to the trend to automation of manufacturing processes and machine control. Collected process data has the potential for optimizing those processes if analyzed in a proper way. But the complex relations, the huge amount of data and the often missing expertise make such optimizations hard to accomplish. Therefore sophisticated methods from the domain of machine learning (ML) and data mining (DM) are often required to identify the relations within the collected data. Furthermore high performance computational hardware is needed to perform these computations in a reasonable amount of time. As most ML algorithms currently only exist in a sequential version, easy transformation into parallel implementations is important as well. Solutions therefore require experts in machine learning for algorithm design and experts in parallelization for implementation on different hardware platforms. Different solutions to deal with the challenge of parallelization of ML algorithms on a higher level of abstraction have been proposed to cope with this problem. One approach is the adaption of the map-reduce (MR) paradigm to execute the algorithms on clusters or multicore machines [28] [58]. But this solution restricts the number of usable ML methods to those that fit the MR paradigm [16]. Other frameworks

like [48] have been published to overcome this restriction, but they do not exploit the potential of heterogeneous shared memory machines. With our use cases we want to demonstrate how the ParaPhrase approach can be used to overcome those shortcomings.

4 Conclusions

This paper has described the newly-started ParaPhrase project. It has introduced key technologies, described the structure of the project, the key related work in the area, and the advances that we anticipate making in the course of the project. ParaPhrase aims to mark a step change in programmability of heterogeneous parallel systems by synthesizing work from several independent areas and by developing new tools and techniques that will allow parallel programmers to develop programs using a tool-supported refactoring approach based on well-understood parallel design patterns coupled with good skeleton implementations. Each of the key technologies that will help achieve this goal is described in depth in a companion paper that has been submitted to this proceedings. Danelutto *et al.* describe a methodology suitable to implement autonomic management of multiple non functional concerns with the patterns and skeletons that we will use in the project; Brown *et al.* describe the refactoring tools and techniques; Gonzalez-Velez *et al.* describe the software virtualisations that we require; and, finally, Aldinucci *et al.* describe the hardware virtualisation layer that underpins the project. Achieving the overall goals of the project represents an exciting technical challenge that will require progress to be made in all these underlying technologies.

Acknowledgements

This work has been supported by the European Union Framework 7 grant IST-2011-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems", http://www.paraphrase-ict.eu.

References

1. S. Adve, V. Adve, G. Agha, M. Frank, et al. Parallel@Illinois: Parallel Computing Research at Illinois — The UPCRC Agenda. http://www.upcrc.illinois.edu/documents/UPCRC_Whitepaper.pdf, Nov. 2008.
2. M. Aldinucci, S. Campa, M. Danelutto, M. Vanneschi, P. Dazzi, D. Laforenza, N. Tonellotto, and P. Kilpatrick. Behavioural skeletons in GCM: autonomic management of grid components. In *Euromicro PDP 2008*, pages 54–63, Toulouse, Feb. 2008. IEEE.
3. M. Aldinucci, M. Danelutto, and P. Dazzi. MUSKEL: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, Dec 2007.
4. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of non-functional concerns in distributed and parallel application programming. In *IPDPS 2009*, pages 1–12, Rome, May 2009. IEEE.

5. M. Aldinucci, M. Danelutto, and P. Kilpatrick. Autonomic management of multiple non-functional concerns in behavioural skeletons. In F. Desprez, V. Getov, T. Priol, and R. Yahyapour, editors, *Grids, P2P and Services Computing*, pages 89–103. Springer-Verlag, 2010.
6. R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2001. ISBN 1-55860-286-0.
7. K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Communications of the ACM*, 52(10):56–67, 2009.
8. D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, 1994.
9. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.
10. R. Bernecky, S. Herhut, S.-B. Scholz, K. Trojahner, C. Grelck, and A. Shafarenko. Index Vector Elimination: Making Index Vectors Affordable. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation and Application of Functional Languages, 18th International Symposium (IFL'06), Budapest, Hungary, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 19–36. Springer-Verlag, Berlin, Heidelberg, New York, 2007.
11. V. Bharadwaj, D. Ghose, V. Mani, and T. G. Robertazzi. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE, Los Alamitos, 1996.
12. U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–113, New York, NY, USA, 2008. ACM.
13. D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia CS*, 1(1):2095–2103, 2010.
14. H. Casanova, M.-H. Kim, J. S. Plank, and J. Dongarra. Adaptive scheduling for task farming with grid middleware. *Int. J. High Perform. Comput. Appl.*, 13(3):231–240, 1999.
15. T. Casavant and J. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, 1988.
16. C.-t. Chu, S. K. Kim, Y.-a. Lin, and A. Y. Ng. Map-reduce for machine learning on multicore. *Architecture*, 19(23):281, 2007.
17. M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, London, 1989.
18. M. Danelutto. On Skeletons and Design Patterns. In G. H. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, editors, *PARALLEL COMPUTING Advances and Current Issues Proceedings of the International Conference ParCo2001*. Imperial College Press, 2002. ISBN:1860943152.
19. M. Danelutto. HPC the easy way: new technologies for high performance application development and deployment. *Journal of Systems Architecture*, 49(10-11):399–419, 2003.
20. D. Dig. A refactoring approach to parallelism. *IEEE Softw.*, 28:17–22, January 2011.
21. H. El-Rewini, T. G. Lewis, and H. H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Innovative Technology Series. Prentice Hall, New Jersey, 1994.

22. P. Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, 1992.
23. P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, pages 79–103, London, UK, 1996. Springer-Verlag.
24. E. Foundation. Eclipse - an Open Development Platform. `http://www.eclipse.org`, 2009.
25. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, July 1993. Springer.
26. E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, Upper Saddle River, 1995.
27. L. Geppert and T. S. Perry. Transmeta's magic show [microprocessor chips]. *IEEE Spectrum,*, 2000.
28. D. Gillick, A. Faria, and J. DeNero. Mapreduce: Distributed computing for machine learning. Berkley, 2006/12/18.
29. H. González-Vélez and M. Cole. An adaptive parallel pipeline pattern for grids. In *IPDPS'08*, pages 1–11, Miami, USA, Apr. 2008. IEEE.
30. H. González-Vélez and M. Cole. Adaptive statistical scheduling of divisible workloads in heterogeneous systems. *Journal of Scheduling*, 13(4):427–441, Aug. 2010.
31. H. González-Vélez and M. Cole. Adaptive structured parallelism for distributed heterogeneous architectures: A methodological approach with pipelines and farms. *Concurrency and Computation–Practice & Experience*, 22(15):2073–2094, Oct. 2010.
32. H. González-Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Software–Practice & Experience*, 40(12):1135–1160, 2010.
33. C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.
34. C. Grelck, K. Hinckfuß, and S.-B. Scholz. With-Loop Fusion for Data Locality and Parallelism. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05), Dublin, Ireland, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 178–195. Springer-Verlag, Berlin, Heidelberg, New York, 2006.
35. C. Grelck, S. Kuthe, and S.-B. Scholz. A Hybrid Shared Memory Execution Model for a Data Parallel Language with I/O. *Parallel Processing Letters*, 18(1):23–37, 2008.
36. C. Grelck, S.-B. Scholz, and A. Shafarenko. A Binding Scope Analysis for Generic Programs on Arrays. In A. Butterfield, editor, *Implementation and Application of Functional Languages, 17th International Workshop (IFL'05). Dublin, Ireland, September 19–21, 2005, Revised Selected Papers*, volume 4015 of *Lecture Notes in Computer Science*, pages 212–230. Springer-Verlag, Berlin, Heidelberg, New York, 2006.
37. C. Grelck, S.-B. Scholz, and K. Trojahner. With-Loop Scalarization: Merging Nested Array Operations. In P. Trinder and G. Michaelson, editors, *Implementation of Functional Languages, 15th International Workshop (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers*, volume 3145 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 2004.
38. M. Griebl. *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. habilitation thesis.

39. S. Herhut, S.-B. Scholz, R. Bernecky, C. Grelck, and K. Trojahner. From Contracts Towards Dependent Types: Proofs by Partial Evaluation. In O. Chitil, Z. Horváth, and V. Zsók, editors, *19th International Symposium on Implementation and Application of Functional Languages (IFL'07), Freiburg, Germany, Revised Selected Papers*, volume (accepted) of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 2008.
40. K. Kennedy, K. S. McKinley, and C. W. Tseng. Interactive parallel programming using the parascope editor. *IEEE Trans. Parallel Distrib. Syst.*, 2:329–341, July 1991.
41. C. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, SE-11(10):1001–1016, Oct. 1985.
42. C. Lengauer. Loop parallelization in the polytope model. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 398–416, London, UK, 1993. Springer-Verlag.
43. P. Leonard. The Multi-Core Dilemma, Intel Software Blog, http://software.intel.com/en-us/blogs/2007/03/14/the-multi-core-dilemma-by-patrick-leonard/. http://software.intel.com/en-us/blogs/2007/03/14/the-multi-core-dilemma-by-patrick-leonard/, Mar. 2007.
44. A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms - extended journal version parallel computing. *Parallel Computing*, 1998.
45. A. W. Lim, S.-W. Liao, and M. S. Lam. Blocking and array contraction across arbitrarily nested loops using affine partitioning. In *PPoPP '01: Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming*, pages 103–112, New York, NY, USA, 2001. ACM.
46. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.
47. S. Long and M. O'Boyle. Adaptive java optimisation using instance-based learning. In *ICS '04: Proceedings of the 18th annual international conference on Supercomputing*, pages 237–246, New York, NY, USA, 2004. ACM.
48. Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. *CoRR*, abs/1006.4990, 2010.
49. S. Majumdar, D. L. Eager, and R. B. Bunt. Scheduling in multiprogrammed parallel systems. *SIGMETRICS Perform. Eval. Rev.*, 16(1):104–113, 1988.
50. T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Software Patterns Series. Addison-Wesley, Boston, 2004.
51. W. F. Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, UIUC, Champaign, IL, USA, 1992.
52. J. Owens, M. Houston, D. Luebke, S. Green, J. Stone, and J. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879 –899, May 2008.
53. A. Pop, S. Pop, and J. Sjödin. Automatic streamization in GCC. In *Proc. of the 2009 GCC Developers Summit*, Montréal, Canada, June 2009.
54. L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: part ii, multidimensional time. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 90–100, New York, NY, USA, 2008. ACM.
55. F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, London, 2003.

56. S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In C. Clack, T. Davie, and K. Hammond, editors, *Implementation of Functional Languages, 9th International Workshop (IFL'97), St. Andrews, Scotland, UK, Selected Papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 72–92. Springer-Verlag, Berlin, Germany, 1998.
57. G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *HCW'00*, pages 3–16, Cancun, May 2000. IEEE.
58. H. Tamano, S. Nakadai, and T. Araki. Optimizing multiple machine learning jobs on mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 59 –66, 29 2011-dec. 1 2011.
59. K. Trojahner, C. Grelck, and S.-B. Scholz. On Optimising Shape-Generic Array Programs using Symbolic Structural Information. In Z. Horváth and V. Zsók, editors, *Implementation and Application of Functional Languages, 18th International Symposium (IFL'06), Budapest, Hungary, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, Berlin, Heidelberg, New York, 2007.
60. U. Mannheim, U. Tennessee and NERSC. TOP500 supercomputer sites. Web site, Nov. 2010. http://www.top500.org/ (Last accessed: 1 Dec 2010).
61. S. S. Vadhiyar and J. Dongarra. Self adaptivity in grid computing. *Concurr. Comput.-Pract. Exp.*, 17(2-4):235–257, 2005.
62. J. Wloka, M. Sridharan, and F. Tip. Refactoring for reentrancy. In *ESEC/FSE '09*, pages 173–182, Amsterdam, 2009. ACM.
63. M. Wolfe. *High-Performance Compilers for Parallel Computing*. Addison-Wesley, 1995. ISBN 0-8053-2730-4.